April 1, 1992

High-Performance Scientific Computing Using C++

K. G. Budge, J. S. Peery and A. C. Robinson
Computational Physics Research and Development (1431)
Sandia National Laboratories
Albuquerque, NM 87185-5800
kgbudge@sandia.gov, jspeery@cs.sandia.gov, acrobin@cs.sandia.gov

Abstract

Concepts from mathematics and physics often map well to object-oriented software since the original concepts are of an abstract nature. We describe our experiences with developing high-performance shock-wave physics simulation codes in C++ and discuss the software engineering issues which we have encountered. The primary enabling technology in C++ for allowed us to share software between our development groups is operator overloading for a number of "numeric" objects. Unfortunately, this enabling feature can also impact the efficiency of our computations. We describe the techniques we have utilized for minimizing this difficulty.

Introduction

Developers of scientific software systems are tasked to implement abstract ideas and concepts. The software implementation of algorithms and ideas from physics, mechanics and mathematics should in principle be complementary to the mathematical abstractions. Often these ideas are very naturally implemented in an object-oriented style. For example, our group is developing software to solve the equations

$$\nabla \bullet \mathbf{T} + \rho \vec{b} = \rho \frac{d^2}{dt^2} \vec{u} \tag{1}$$

$$\frac{d}{dt}\mathbf{T} = f(\nabla \frac{d}{dt}\vec{u}, \mathbf{T}, \ldots) \tag{2}$$

subject to a variety of boundary conditions. These equations describe the mechanical response of a continuous medium. They contain numerous abstract mathematical objects such as scalar, vector, and tensor fields, arithmetic operators, and calculus operators. Physical concepts such as the equation of state of a physical substance and its constitutive response model can also be encapsulated very cleanly as a object. Similar objects are found in the equations describing many other physical systems. All of these can be represented by appropriate data structures in a high-level computer language.

Traditionally, scientists have relied on FORTRAN for high-performance computing. The reasons for this are clear. FORTRAN optimizing compilers have been around for a long time and produce extremely efficient code. Large libraries of numerical routines are available in the language. It is familiar to almost every scientific worker. The drawbacks of standard FORTRAN are also well-known. It is devoid of mandated type checking. It has no support for structures. The concept of a free memory store does not exist. Its grammar encourages an atrocious style of programming. Note that these statements are true of *standard* FORTRAN-77. The deficiencies of FORTRAN are so painful that numerous vendors have provided proprietary extensions to the language. For example, most vendors now support the nonstandard DO-ENDDO construct. Some codes simply cannot avoid using FORTRAN language extensions. For example, codes using large databases and complicated algorithms *must* find a way to dynamically allocate memory. Since no provisions

for this exists in standard FORTRAN-77, programmers must either use proprietary extensions which supply the existing functionality, make calls to system level routines based on C or develop intricate memory management schemes from common memory.

It is not unknown for a large scientific production code written in FORTRAN to exceed half a million lines in length and to include numerous platform-dependent statements. This represents a maintenance challenge comparable to that for a small operating system. Furthermore, such code tends to be very opaque. Unless the code is extremely well-documented, the transfer or retirement of one of its programmers can effectively freeze portions of the code, because no one else will ever be able to figure out what it does.

Many scientific programming groups have come to the realization that such programming practices are prohibitively expensive. What is needed is a programming environment in which code is highly reusable, transparent to the reader, and easily debugged during development and maintenance, but which retains FORTRAN-like efficiency. As a result, there is a growing interest in more modern languages in the scientific community. C++ has attracted the most interest because of its wide availability and support in the general programming community and because of its explicit design goals to provide object-oriented functionality and excellent software engineering characteristics in a way which does not totally destroy efficiency. Applications include distributed particle simulations, partial differential equations, fluid mechanics, robotic languages, mesh generation, adaptive grid methods, data-parallel C, general numeric libraries, image algebras, large scientific database management and genetic algorithms. See references [1]-[20] which are organized roughly chronologically.

It should be pointed out that the Fortran 90 standard has recently been adopted [21]. In addition a specifically non-ANSI based effort is underway to agree upon a High Performance Fortran standard this year. The final universal acceptance and wide availability of compilers for these languages is likely but on an uncertain time scale. A major reason for not choosing to do our current development work in a Fortran 90 style language was the unclear state of the standard at the time we began our work as well as very uncertain availability and support considerations on our rapidly changing target architectures. In addition, current language specifications do not appear to provide the kind of robust software engineering characteristics which we have come to enjoy.

In this paper we briefly describe our use of C++ in two large-scale scientific code development projects and give examples of how we have mapped interesting physical, mathematical and computational concepts to C++. One project is developing a shock wave physics simulation code (PCTH) for fixed “Eulerian” grids with massively parallel MIMD architectures as the primary target, and the other project (RHALE++) uses an “Arbitrary Lagrangian-Eulerian” technology based on an unstructured grid finite element technology. In the following sections we describe several object-oriented concepts which we are utilizing and how we are attempting to realize the power of the abstraction capability and the excellent software engineering features which the C++ language provides. Subsequently, we discuss some aspects of the efficiency difficulties which we have encountered as well as approaches to their resolution.

C++ As a Meta-Language for Mathematical Physics

We tend to regard C++ as being essentially a *meta-language* whose dialects can be tailored to a particular field of application. Our dialects of C++ are tailored to mathematical physics. For example, the polar decomposition of a velocity gradient $\mathbf{L}$ is expressed by the equations [22]

$$\mathbf{D} = \frac{1}{2}(\mathbf{L} + \mathbf{L}^T) = \mathrm{Sym}(\mathbf{L}) \tag{3}$$

$$\mathbf{W} = \frac{1}{2}(\mathbf{L} - \mathbf{L}^T) = \mathrm{Anti}(\mathbf{L}) \tag{4}$$

$$\vec{z} = [\varepsilon_{ijk} V_{jm} D_{mk}] = \mathrm{Dual}(\mathbf{VD}) \tag{5}$$

$$\vec{\omega} = \mathrm{Dual}(\mathbf{W}) - 2(\mathbf{V} - \mathrm{Tr}(\mathbf{V})\mathbf{1})^{-1}\vec{z} \tag{6}$$

$$\Omega = \frac{1}{2}\mathrm{Dual}(\vec{\omega}) \tag{7}$$

$$\frac{d\mathbf{R}}{dt} = \Omega\mathbf{R} \tag{8}$$

$$\frac{d\mathbf{V}}{dt} = \mathbf{L}\mathbf{V} - \mathbf{V}\Omega \tag{9}$$

The time discretization used to integrate the last two equations is

$$\mathbf{R}^{n+1} = \left(\mathbf{1} - \frac{1}{2}(\Delta t)\Omega\right)^{-1}\left(\mathbf{1} + \frac{1}{2}(\Delta t)\Omega\right)\mathbf{R}^{n} \tag{10}$$

$$\mathbf{V}^{n+1} = \mathbf{V}^{n} + \mathrm{Sym}(\mathbf{L}\mathbf{V} - \mathbf{V}\Omega)\Delta t \tag{11}$$

In RHALE++ we have defined classes representing the vector and tensor objects in these equations [23]. Using these classes, we can code this algorithm as

```
void Decompose(const double delt, SymTensor& V,
  Tensor& R,  const Tensor& L)
{
  SymTensor D;
  AntiTensor W, Omega;
  Vector z, omega;

  D = Sym(L);
  W = Anti(L);

  z = Dual(V*D);
  omega = Dual(W) - 2.0 * Inverse(V - Tr(V) * One) * z;
  Omega = 0.5 * Dual(omega);

  R = Inverse(One - 0.5 * delt * Omega) *
           (One + 0.5 * delt * Omega) * R;
  V += delt * Sym(L * V - V * Omega);
}
```

Note the heavy use of operator overloading. This code is transparent and its underlying class libraries are versatile and easy to maintain. A physicist familiar with the polar decomposition algorithm can make immediate sense of this code fragment without the need for any additional documentation.

By contrast, the FORTRAN version of this subroutine is

```
 subroutine decompose(delt,V_xx,V_xy,V_xz, V_yy,
* V_yz, V_zz, R_xx, R_xy, R_xz, R_yx, R_yy, R_yz,
* R_zx, R_zy, R_zz, L_xx, L_xy, L_xz, L_yx, L_yy,
* L_yz, L_zx, L_zy, L_zz)
 D_xx = L_xx
 D_xy = 0.5*(L_xy+L_yx)
```

... about three pages of these proceedings . . .

```
V_xy = V_xy + 0.5*(t3_xy + t3_yx)
V_zz = V_zz + t3_zz
 return
 end
```

The stylistic advantages of C++ are obvious. The second subroutine (in FORTRAN) is virtually unreadable. It is also very difficult to debug. However, the FORTRAN version is somewhat more efficient. Several expressions are evaluated in the C++ version that are never used. In principle, a sufficiently intelligent optimizer could eliminate these expressions.

Field classes representing scalar, vector, and tensor fields are fundamental to our approach to simulation coding. Field classes are coded as sets of smart arrays representing the components of vector or tensor fields. At this level, no topological information is included in the fields. Thus, while numerous element-by-element operations are overloaded, no calculus operations are defined. These are added in classes derived from the basic field classes.

These field classes hide subscripting and loops, eliminating a common source of error in FORTRAN code. In the FORTRAN code fragment above the functional interface looks essentially the same whether the arguments are scalars or data-parallel arrays. In the C++ case the same holds true except the compactness of the high-order tensor fields hides the extent of the implied data and the interactions between tensor elements implied by the mathematical operations.

Encapsulation of Physical Concepts

In our simulation codes empirical equations are used along with the basic conservation equations to describe how a given material behaves. In shock physics, two of these empirical equations are an equation of state and a constitutive model. An equation of state generally determines the pressure, temperature, and sound speed of a material based on the material's density and energy. A constitutive model, on the other hand, determines the stress state of a material based on the material deformation. Within each of these concepts numerous models are available; however, the inputs and outputs are the same. C++ can be used to encapsulate the uniqueness of a model (particularly the private data) and provide a common interface to the concept.

Using C++ to encapsulate physical concepts can be best seen in an equation of state class. Encapsulation begins with the concept of an abstract class that contains all data and functions common to every equation of state. A simplified summary of such a class is given as

```
class Equation_of_State {
      public:
            Equation_of_State();
            Equation_of_State(const Equation_of_State&);
            ~Equation_of_State();
            virtual void Update_Thermodynamic_State(
                  const Field& density,
                  const Field& energy,
                  Field& pressure,
                  Field& temperature,
                  Field& sound_speed) const = 0;
};
```

This abstract class provides a common interface to updating the thermodynamic state of materials and contains data that is unique to every equation of state. Constructing a unique equation of state simply requires deriving from the abstract class, adding any unique data, and providing a unique function for updating the thermodynamic state of the material. For example, an ideal gas equation of state class is given as

```
class Ideal_Gas : public Equation_of_State {
      private:
            double gamma; // Ratio of specific heats
            double cv; // Specific heat
            double gamma_minus_one; // Gamma - 1
      public:
            Ideal_Gas();
            Ideal_Gas(const Ideal_Gas&);
            ~Ideal_Gas();
            void Update_Thermodynamic_State(
                  const Field& density,
```

```
                        const Field& energy,
                        Field& pressure,
                        Field& temperature,
                        Field& sound_speed) const;
};
```

The third private variable (gamma-1) is indicative of the internal constants which a given equations of state might create in order to more efficiently perform its functions as a server class for fields which are passed to it. With this approach, to add an additional equations of state option one only has to modify the physics code at the one location where a particular type of equation of state is specified for a given material. The rest of the code utilizes only the base class pointer and the correct function are called at run time. In addition, by carefully design, a class that abstracts the physics can be used by many codes (the equation of state and constitutive classes are being designed to be used both by PCTH and RHALE++) and can be extended quickly. For example, it has taken less than 4 hours to add a complicated new constitutive model to the RHALE++ code.

Tracer Particle Objects

PCTH is being developing with massively parallel MIMD architectures as a primary focus although with a data-parallel syntax to facilitate access to vector hardware. One of the requirements for these codes is for something that is called a *tracer particle*. These are points which do not interact with the simulation flow field but are required to follow the flow. Since the PCTH code decomposes work spatially by subdividing the simulation space, these particles (objects) need to skip from node to node of the massively parallel machines according to the dynamics of the flow field. These particles store an identifier, position, velocity and local state values. They know how to send themselves to neighboring nodes as well as how to create a copy of a particle which as been sent from a neighboring node. The physics code knows how to interpolate state values onto the tracer particles.

C++ Software Engineering Experience

At this point in time there are relatively few classes for purchase or in the public domain that address the needs of scientific community and, of these, none have been ported and optimized for vector and parallel computing architectures. In addition, there are no classes for scientific computing that have been adopted as a standard. Part of the philosophy behind C++ is not to “re-invent the wheel;” however, at this time, most of the burden of developing base classes, such as matrix, array, and vector classes, falls on the members of our project teams. As a result, there is a proliferation of scientific class libraries, each with their own particular syntax, functionality, and performance.

The redundance of these scientific classes is very unfortunate because developing robust base classes is extremely expensive. One must hope that the lifetime of the code project is long enough to reap the rewards of the initial investment in developing the base classes.

The RHALE++ and PCTH projects both began with their own flavors of Field classes. The RHALE++ project implemented the field classes as arrays of objects while the PCTH project implemented an object of arrays approach. However, it was realized that although the initial investment of developing a robust array class is expensive, it is equally expensive to port and tune the class to a particular computer architecture. As a result, a common array class, which serves as the building block for scalar, vector, and tensor fields, is being developed as the basis for performing mathematical operation in both codes in an “object of arrays” approach. We feel that this is a wise investment since the performance of the array class is a direct indication of the performance of theses two codes and thus only this class must be tuned for a particular computer architecture and both codes will benefit.

Although developing robust base classes is costly, extending or deriving from these classes is relatively inexpensive. For example, extending an array class to a vector array class would be very inexpensive for all the operations for which one operates on the components of a vector.

A major reason for going with C++ was the complexity of the algorithms which we were dealing with in the case of the RHALE++ project and the complexity of the underlying (current and unknown future) parallel architectures in the case of the PCTH project. By using object-oriented concepts we can hide the underlying architectures to a great degree and thus enable us to quickly port to any machine which we may be required to run our code on.

The fact that C++ is based on C gives us confidence that a C++ environment can be provided relatively cheaply on any architecture of interest. The architectures potentially include a variety of MIMD, SIMD, data-parallel and vector processors as well networks of high performance workstations. In addition we hope to eventually take full advantage of the workstation software tools market even on the exotic architectures which we must deal with. We believe that much of this software will be written in C++ in the future. In fact, in PCTH we are already utilizing a commercial solid modeling toolkit written in C++ to implement the volume fraction computations in the initialization phase of our simulations. As we consider balancing production software development with the uncertainties of the future, C++ has appeared to be an excellent bet. Our rational is similar to common arguments given for moving to C++ for production development [24].

One of the most enticing aspects of C++ is the elimination of many common bugs introduced in Fortran coding. Many of these bugs are removed by the strong type checking and function argument matching capabilities inherent in C++. With the use of overloaded operators, index mistakes are eliminated or reduced to a single location in the code which can be easily identified and corrected. For most scientific applications, ninety percent or more of the bugs have just been eliminated. However, for most Fortran programmers that switch to C or C++, a new and much more nasty bug is potentially introduced with the concept of pointers. Fortunately, there are very nice debugging environments readily available for C++ due to the strong following in the overall programming community.

In the RHALE++ and PCTH projects, it has been our experience that we spend much less time debugging executable C++ code than Fortran code and that most of our debugging activities are in debugging the physics (incorrectly coded or poor selection of algorithms) and sometimes in locating bad pointers. This excellent experience is due mainly to our use of overloaded operators and the robust error detection features of C++ compilers.

C++ Reusability

Another aspect of C++ and object-oriented programming is the ease with which new physical models can be added to a standing code and ported between platforms and projects. Both the PCTH and RHALE++ projects are developing large and complex codes which will be evolving for a number of years. PCTH uses a finite-difference method, whereas RHALE++ uses a finite-element method. These two approaches to numerical calculus differ considerably.

Despite the profound difference in overall methods, we are now in the process of merging code portions which are conceptually identical. This is possible because of the field class concept which is inherent in the underlying physics and thus the codes themselves. We are now converting the two codes to use the same set of basic field classes, which are then specialized for finite difference/finite element field classes. Both codes must eventually run on a wide variety of platforms, ranging from PCs to the newest massively parallel computers. The architectural differences between platforms require different optimization strategies. We are confident that these can be hidden in a great degree in the field class libraries. The need to write efficient code for massively parallel computers and the prospect of hiding domain decomposition and message passing in the field class libraries was a strong incentive to investigate scientific C++ in the first place.

The equations of state and strength models are derived from abstract base classes but take fields as arguments. It does not really matter whether the underlying field is based on field elements or finite differences. This makes it relatively simple to incorporate new equations of state or strength models in the codes.

The original equation of state code was developed for PCTH. It was then moved relatively easily to RHALE and the new upgraded capabilities will move readily back to PCTH. At this time we are converging on the correct specification and approach.

Complicated algorithms can also benefit from abstract formulations. We have had the experience of moving

a complicated interface tracking algorithm from PCTH to RHALE essentially as a cut-and-paste.

The fundamental reason why we are able to move in the direction of better code reuse is because of the common underlying physical concepts which must be modeled by out two codes. The connecting link is the use of field classes which can be considered to be a “numeric” type with appropriate overloaded operators. Unfortunately overloaded operators, which have the potential for such excellent code maintainability, reusability and extensibility for our development projects, are also the major impediment for efficient execution of our codes. These difficulties and the extent to which they have been solved will now be described.

C++ Efficiency Issues for Overloaded Operators

All the advantages of C++ are unfortunately offset to some degree by the difficulty of developing efficient overloaded operator methods for large array objects. Many high performance machines require array (vector) operations to achieve good performance. The fact that the current C++ language and implementations do not support compiler optimization of large arithmetic objects, as if they were fundamental types, causes extensive difficulties for developing efficient C++ applications.

This is illustrated in Table 1, which shows the estimated processor speeds for test versions of the polar decomposition algorithm. Both C++ and FORTRAN-77 versions were tested on Sun workstations and Sandia’s CRAY-YMP for a range of field sizes. The C++ version uses reference counting and memory management techniques discussed later in this section. In addition, innermost loops are implemented through calls to FORTRAN subroutines (using the C++ external linkage specification facility). The statistics show that peak speed for pure FORTRAN-77 is two or three times greater than that for C++ and that C++ involves a considerable overhead (reflected in the array size at which one gets 50% of peak performance).

Table 1. Performance of Field Classes In a Polar Decomposition Test Problem

# elements	Sun Sparc-2 MFlops	Sun Sparc-2/F MFlops	Cray YMP MFlops	Cray YMP/F MFlops
1	0.04	--	0.11	12.7
10	0.31	1.3	1.03	63.7
100	0.92	3.1	9.6	185.8
1000	0.89	2.7	59.1	221.9
10000	0.88	2.7	123.7	225.4
100000	out of memory	out of memory	140.0	225.9
50% peak	18 elements	13 elements	1380 elements	27 elements

These figures are somewhat discouraging taken at face value. The Cray is a vector machine and its overall performance is extremely sensitive to scalar portions of the code. The scalar overhead tends to be dominant except when vector lengths are large. The peak speed of C++ is about half of the Fortran speed.

What accounts for these results? The problem is that C++ strongly isolates binary operations. Consider the following test program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

class Array {
```

```
 public:
 Array(void);
 Array(const int n);
 Array(const int n, const double[]);
 Array(const Array&);
 ~Array(void);
 Array& operator=(const Array&);

 double& operator[](const int);

 friend Array operator*(const Array&, const Array&);
 friend Array operator+(const Array&, const Array&);

 private:
 int n;
 double *data;
};

inline Array::Array(void) :
 n(0),
 data(NULL)
{}

inline Array::Array(const int nn) :
 n(nn),
 data(new double[nn])
{}

inline Array::Array(const int nn, const double d[]) :
 n(nn),
 data(new double[nn])
{
 memcpy(data, d, sizeof(double)*n);
}

inline Array::Array(const Array& src) :
 n(src.n),
 data(new double[src.n])
{
 memcpy(data, src.data, sizeof(double)*n);
}

inline Array::~Array(void){
 delete data;
}

inline Array& Array::operator=(const Array& src){
 delete data;
 n = src.n;
 data = new double[n];
 memcpy(data, src.data, sizeof(double)*n);
 return *this;
}

inline double& Array::operator[](const int nn){
 if (nn<0 || nn > n) abort();
 return data[nn];
}
```

```
inline Array operator*(const Array& a, const Array& b){
 Array result(a.n);
 for (register i=0; i<a.n; i++)
 result.data[i] = a.data[i] * b.data[i];
 return result;
}

inline Array operator+(const Array& a, const Array& b){
 Array result(a.n);
 for (register i=0; i<a.n; i++)
 result.data[i] = a.data[i] + b.data[i];
 return result;
}

main(){
const int SIZE = 50000;
 static double a_data[SIZE] = { 2., 5., 3., 8. };
 static double b_data[SIZE] = { 5., 3., 4., 2. };
 static double c_data[SIZE] = { 3., 4., 5., 6. };
 static double x_data[SIZE] = { 2., 5., 7., 9.};

 Array A(SIZE, a_data);
 Array B(SIZE, b_data);
 Array C(SIZE, c_data);
 Array X(SIZE, x_data);
 Array Y;

 for (register i=0; i<100; i++){
 Y = C + X*(B + X*A);
 }

 printf("First element is %f\n", Y[0]);
}
```

The C and FORTRAN equivalents of this program achieve a peak performance of 230 MFlops on the CRAY-YMP. This unoptimized C++ version achieves only 49 MFlops.

The CFRONT translater generates code which is essentially equivalent to

```
struct Array {
 int n ;
 double *data ;
};

static char multiply_Array_Array(
 struct Array *result,
 struct Array *a,
 struct Array *b )
{
 struct Array Result ;
 register int i ;
 int itmp ;

 itmp = a->n;
 Result.n = itmp;
 Result.data = (double*)malloc(sizeof(double)* itmp);

 for(i=0;i < a-> n ;i ++ )
 Result.data[i] = a->data [i] * b->data [i];
```

```
 result->n = Result.n;
 result->data =
(double*)malloc(sizeof(double)*Result.n);
 memcpy(result->data, Result.data,
 sizeof(double) * result->n);

 free (Result.data);
 return ;
}

static char add_Array_Array (
 struct Array *result,
 struct Array *a,
 struct Array *b )
{
 struct Array Result ;
 register int i ;
 int itmp ;

 itmp = a->n;
 Result.n = itmp;
 Result.data = (double *)malloc(sizeof(double)*itmp);

 for(i=0;i < a->n ;i ++ )
 Result.data[i] = a->data[i] + b->data[i];

 result->n = Result.n;
 result->data =
(double *)malloc(sizeof(double)* Result.n);
 memcpy(result->data, Result.data ,
 sizeof(double)* result->n);

 free (Result.data );
 return ;
}

int main (){

 static double a_data [50000]= { 2. , 5. , 3. , 8. } ;
 static double b_data [50000]= { 5. , 3. , 4. , 2. } ;
 static double c_data [50000]= { 3. , 4. , 5. , 6. } ;
 static double x_data [50000]= { 2. , 5. , 7. , 9. } ;

 struct Array A ;
 struct Array B ;
 struct Array C ;
 struct Array X ;
 struct Array Y ;

 register int i ;

 A.n = 50000;
 A.data = (double *)malloc (sizeof(double)* 50000);
 memcpy(A.data , (double*)a_data, sizeof(double)* A.n);

 B.n = 50000;
 B.data = (double *)malloc (sizeof(double)* 50000);
 memcpy(B.data , (double*)b_data, sizeof(double)* B.n);
```

```
 C.n = 50000;
 C.data = (double *)malloc (sizeof(double)* 50000);
 memcpy(C.data , (double*)c_data, sizeof(double)* C.n);

 X.n = 50000;
 X.data = (double *)malloc (sizeof(double)* 50000);
 memcpy(X.data , (double*)x_data, sizeof(double)* X.n);

 for(i=0;i < 100 ;i ++ ) {
 struct Array atmp1 ;
 struct Array atmp2 ;
 struct Array atmp3 ;
 struct Array atmp4 ;

 multiply_Array_Array(&atmp1, &X, &A);
 add_Array_Array(&atmp2, &B, &atmp1);
 multiply_Array_Array(&atmp3 , &X, &atmp2);
 add_Array_Array(&atmp4, &C, &atmp3);

 free(Y.data);
 Y.n = atmp4.n;
 Y.data = (double *)malloc(sizeof(double)* Y.n);
 memcpy(Y.data, atmp4.data, sizeof(double)* Y.n);


 free(atmp4.data);
 free(atmp3.data);
 free(atmp2.data);
 free(atmp1.data);
 }

 printf ("First element is %f\n", (0 > Y.n ? abort(),
 0 : Y.data[0]));\

 free(Y.data);
 free(X.data);
 free(C.data);
 free(B.data);
 free(A.data);
}
```

Loops are not inlined. Thus, although they vectorize individually, they cannot be chained. In addition, considerable effort is wasted allocating and de-allocating memory for the temporaries and copying the contents of temporaries to local variables.

The latter difficulty is solvable through the well-known technique of *reference counting*. Using this technique, the definition of `class Array` is changed as follows:

```
class Array {
 public:
 Array(void);
 Array(const int n);
 Array(const int n, const double[]);
 Array(const Array&);
 ~Array(void);
 Array& operator=(const Array&);

 double& operator[](const int);
```

```
 friend Array operator*(const Array&, const Array&);
 friend Array operator+(const Array&, const Array&);

 private:
 int n;
 int *ref_count;
 double *data;
};

inline Array::Array(void) :
 n(0),
 data(NULL),
 ref_count(NULL)
{}

inline Array::Array(const int nn) :
 n(nn),
 ref_count(new int(1)),
 data(new double[nn])
{}

inline Array::Array(const int nn, const double d[]) :
 n(nn),
 ref_count(new int(1)),
 data(new double[nn])
{
 memcpy(data, d, sizeof(double)*n);
}

inline Array::Array(const Array& src) :
 n(src.n),
 data(src.data),
 ref_count(src.ref_count)
{
 if (ref_count) (*ref_count)++;
}

inline Array::~Array(void){
 if (ref_count && !--*ref_count){
 delete data;
 delete ref_count;
 }
}

inline Array& Array::operator=(const Array& src){
 if (ref_count && !--*ref_count){
 delete ref_count;
 delete data;
 }
 n = src.n;
 ref_count = src.ref_count;
 if (ref_count) (*ref_count)++;
 data = src.data;
 return *this;
}
```

With this change, the peak computation rate jumps to 84 MFlops -- a 70% increase, but still far short of the 230 MFlops achieved with conventional C coding.

Profiling of our test code shows that, for moderate array sizes, much time is spent in the memory allocation and deallocation routines. Our experience is that the scalar overhead can be significantly reduced by taking charge of memory management through overloaded `new` and `delete` operators. If we use the memory manager we developed for the field class library, the computation rate jumps from 3.7 to 8.5 MFlops for small arrays (~128 elements)..

Now consider some hypothetical future enhancements to the compiler. If loops inlined, the translator would produce code within the main program loop equivalent to

```
for(i=0; i < 100 ; i ++ ) {
  struct Array atmp1 ;
  struct Array atmp2 ;
  struct Array atmp3 ;
  struct Array atmp4 ;
  struct Array Result ;
  register int ii;

  Result.n = X.n;
  Result.data = (double *)malloc(sizeof(double)*X.n);
  Result.ref_count = (int *)malloc(sizeof(int));
  *Result.ref_count = 1;

  for(ii=0; ii < X.n; ii++ )
    Result.data[ii] = X.data[ii] * A.data[ii];

  atmp1.n = Result.n;
  atmp1.data = Result.data;
  atmp1.ref_count = Result.ref_count;
  if (atmp1.ref_count) (*atmp1.ref_count)++;

  if (Result.ref_count && ! --(*Result.ref_count)){
    free (Result.data);
    free (Result.ref_count);
 }

  Result.n = B.n;
  Result.data = (double *)malloc(sizeof(double)*B.n);
  Result.ref_count = (int *)malloc(sizeof(int));
  *Result.ref_count = 1;

  for(ii=0; ii < B.n; ii++ )
    Result.data[ii] = B.data[ii] + atmp1.data[ii];

  atmp2.n = Result.n;
  atmp2.data = Result.data;
  atmp2.ref_count = Result.ref_count;
  if (atmp2.ref_count) (*atmp2.ref_count)++;

  if (Result.ref_count && ! --(*Result.ref_count)){
    free (Result.data);
    free (Result.ref_count);
 }

  Result.n = X.n;
  Result.data = (double *)malloc(sizeof(double)*X.n);
  Result.ref_count = (int *)malloc(sizeof(int));
  *Result.ref_count = 1;

  for(ii=0; ii < X.n; ii++ )
```

```
    Result.data[ii] = X.data[ii] * atmp2.data[ii];

  atmp3.n = Result.n;
  atmp3.data = Result.data;
  atmp3.ref_count = Result.ref_count;
  if (atmp3.ref_count) (*atmp3.ref_count)++;

  if (Result.ref_count && ! --(*Result.ref_count)){
    free (Result.data);
    free (Result.ref_count);
  }

  Result.n = C.n;
  Result.data = (double *)malloc(sizeof(double)*C.n);
  Result.ref_count = (int *)malloc(sizeof(int));
  *Result.ref_count = 1;

  for(ii=0; ii < C.n; ii++ )
    Result.data[ii] = C.data[ii] + atmp3.data[ii];

  atmp4.n = Result.n;
  atmp4.data = Result.data;
  atmp4.ref_count = Result.ref_count;
  if (atmp4.ref_count) (*atmp4.ref_count)++;

  if (Result.ref_count && ! --(*Result.ref_count)){
    free (Result.data);
    free (Result.ref_count);
  }

  if (Y.ref_count && !--(*Y.ref_count)){
    free(Y.ref_count);
    free(Y.data);
  }
  Y.n = atmp4.n;
  Y.ref_count = atmp4.ref_count;
  if (Y.ref_count) (*Y.ref_count)++;
  Y.data = atmp4.data;

  if (atmp4.ref_count && !--(*atmp4.ref_count)){
    free(atmp4.ref_count);
    free(atmp4.data);
  }
  if (atmp3.ref_count && !--(*atmp3.ref_count)){
    free(atmp3.ref_count);
    free(atmp3.data);
  }
  if (atmp2.ref_count && !--(*atmp2.ref_count)){
    free(atmp2.ref_count);
    free(atmp2.data);
  }
  if (atmp1.ref_count && !--(*atmp1.ref_count)){
    free(atmp1.ref_count);
    free(atmp1.data);
  }
}
```

The individual loops are separated by substantial sections of code. Close examination of these reveals that the code can be rearranged:

```
for(i=0; i < 100 ; i ++ ) {
  struct Array atmp1 ;
  struct Array atmp2 ;
  struct Array atmp3 ;
  struct Array atmp4 ;
  struct Array Result ;
  register int ii;

  Result.n = X.n;
  Result.data = (double *)malloc(sizeof(double)*X.n);
  Result.ref_count = (int *)malloc(sizeof(int));
  *Result.ref_count = 1;

  atmp1.n = Result.n;
  atmp1.data = Result.data;
  atmp1.ref_count = Result.ref_count;
  if (atmp1.ref_count) (*atmp1.ref_count)++;

  if (Result.ref_count && ! --(*Result.ref_count)){
    free (Result.data);
    free (Result.ref_count);
 }

  Result.n = B.n;
  Result.data = (double *)malloc(sizeof(double)*B.n);
  Result.ref_count = (int *)malloc(sizeof(int));
  *Result.ref_count = 1;

  atmp2.n = Result.n;
  atmp2.data = Result.data;
  atmp2.ref_count = Result.ref_count;
  if (atmp2.ref_count) (*atmp2.ref_count)++;

  if (Result.ref_count && ! --(*Result.ref_count)){
    free (Result.data);
    free (Result.ref_count);
 }

  Result.n = X.n;
  Result.data = (double *)malloc(sizeof(double)*X.n);
  Result.ref_count = (int *)malloc(sizeof(int));
  *Result.ref_count = 1;

  atmp3.n = Result.n;
  atmp3.data = Result.data;
  atmp3.ref_count = Result.ref_count;
  if (atmp3.ref_count) (*atmp3.ref_count)++;

  if (Result.ref_count && ! --(*Result.ref_count)){
    free (Result.data);
    free (Result.ref_count);
  }

  Result.n = C.n;
  Result.data = (double *)malloc(sizeof(double)*C.n);
  Result.ref_count = (int *)malloc(sizeof(int));
  *Result.ref_count = 1;

  atmp4.n = Result.n;
```

```
  atmp4.data = Result.data;
  atmp4.ref_count = Result.ref_count;
  if (atmp4.ref_count) (*atmp4.ref_count)++;

  if (Result.ref_count && ! --(*Result.ref_count)){
    free (Result.data);
    free (Result.ref_count);
  }

  if (Y.ref_count && !--(*Y.ref_count)){
    free(Y.ref_count);
    free(Y.data);
  }
  Y.n = atmp4.n;
  Y.ref_count = atmp4.ref_count;
  if (Y.ref_count) (*Y.ref_count)++;
  Y.data = atmp4.data;

  for(ii=0; ii < X.n; ii++ )
    atmp1.data[ii] = X.data[ii] * A.data[ii];

  for(ii=0; ii < B.n; ii++ )
    atmp2.data[ii] = B.data[ii] + atmp1.data[ii];

  for(ii=0; ii < X.n; ii++ )
    atmp3.data[ii] = X.data[ii] * atmp2.data[ii];

  for(ii=0; ii < C.n; ii++ )
    atmp4.data[ii] = C.data[ii] + atmp3.data[ii];

  if (atmp4.ref_count && !--(*atmp4.ref_count)){
    free(atmp4.ref_count);
    free(atmp4.data);
  }
  if (atmp3.ref_count && !--(*atmp3.ref_count)){
    free(atmp3.ref_count);
    free(atmp3.data);
  }
  if (atmp2.ref_count && !--(*atmp2.ref_count)){
    free(atmp2.ref_count);
    free(atmp2.data);
  }
  if (atmp1.ref_count && !--(*atmp1.ref_count)){
    free(atmp1.ref_count);
    free(atmp1.data);
  }
}
```

This puts the loops together and permits chaining. The only assumptions made in rearranging the code in this manner are that *the allocation/deallocation routines have no side effects* and that *the allocation routine returns a pointer to unaliased memory.* This is equivalent to requiring that the global operators `new` and `delete` have no side effects and no aliasing. Failure to obtain memory in the allocation routine should throw an exception rather than returning a null pointer. If permitted to make these assumptions, an extremely intelligent compiler might eliminate the allocation/deallocation operations entirely.

Several other techniques for improving vector performance are known, although we have not implemented them in our present codes. The technique of *deferred expression evaluation* eliminates nearly all large temporaries, but with a significant overhead cost. The technique consists of building a parse tree for each expres-

sion at run time, which is only evaluated when it is assigned to a variable or otherwise used. Temporaries contain tree nodes rather than data and use a relatively small amount of memory. Furthermore, one can apply optimizations to the parse tree, although the run time overhead involved may be prohibitive unless the arrays are very large [25].

An equivalent effect (with greatly reduced overhead) could be obtained by extending the C++ language to permit overloading of entire parse trees. For example, the signature

```
Array& operator=+(Array&, const Array&, const Array&)
```

might correspond to

```
a = b + c;
```

while

```
Array operator+*(const Array&, const Array&, const Array&)
```

would correspond to

```
a + b*c;
```

The chief objection to this approach is its complexity, particularly if one attempts to extend it to arbitrarily complex parse trees.

Another approach would be to permit users to specify optimizations along with the definition of a class and its operations. For example, one could instruct the compiler to replace all expressions of the form

```
a = b + c*d;
```

with the expression

```
a = c*d, a += b;
```

which might be easier to optimize because no memory allocations take place between the evaluation of the two sub-expressions.

Although we have no direct experience to date the advent of return value optimizing compilers is encouraging [26].

The simplest solution may be to standardize an array class. Since the array class name would become a reserved identifier, vendors would be free to develop compilers that implement the array class as a built-in type.

Even with the difficulties described above, we have seen that our C++ code can perform well with present C++ language systems. We have simulated impact events on several high performance computer architectures using our C++ codes, and have obtained performance results which are competitive with previous generation FORTRAN codes. In the case of PCTH running on the nCUBE hypercube we have implemented imbedded assembly language routines for many of the operations in the base field classes (though not yet in the calculus type operator classes). Results to date show that for sufficiently large granularities no more than about a 50 percent loss will be sustained. Further optimization should improve this estimate considerably. The required granularity tends to be much larger than the granularity required for good efficiencies due to message passing overhead but not so large as to be unrealistic for utilizing the machine effectively for our simulations. On 1 CPU of the Cray YMP, the PCTH code has achieved 90 percent of the original CTH (FORTRAN 77) code speed on a fairly complicated two-dimensional problem with a 250x250 field granularity. In this case as well, smaller problems suffer from scalar code overhead as would be expected from the results described earlier. The reasons the numbers compare so well for the CRAY is the fact that every single vector operation in the field classes have been carefully optimized and the fundamental vector lengths turn out to be much larger in the PCTH code than in the CTH code (due to the field abstraction). It is also possible that there are not sufficient chaining operations in the CTH algorithms to significantly boost overall performance over the binary operation limit. These results, for a base field class whose methods implement reference counting, internal memory management and specialized routines, are sufficiently encouraging for us to believe that C++ can become an extremely effective language for scientific and engineering programming as better class libraries, language features, and optimizing compilers become generally available.

Summary

The abstractions and software engineering properties of the C++ language have been found to be an excellent fit to large scale scientific software development for the strong shock wave physics codes which we are developing. We have found that considerable effort must be expended in the design (and redesign) of base classes which are to be used in our codes. However, once these classes are developed, we find that excellent control over the development of additional code is obtained. We have demonstrated that code based on object-oriented ideas can be more readily reusable (shareable) by other scientific developers. We plan to share our code to an even greater extent in the future as current classes are redesigned. Good reusability and shareability for our application classes and algorithms tends to hinge on the extensive use of operator overloading for objects which are essentially "numeric" types. Unfortunately, this is also the most difficult aspect of the language to implement with efficiencies which approach C or Fortran. The various techniques which we have implemented to improve operator overloading efficiencies allow us to approach but not exceed Fortran efficiencies on the high-performance architectures which we have investigated as long as our objects are of sufficiently large granularity. Our current efficiency estimates are within acceptable limits but we consider that much more attention must be paid to issues of optimization of numeric types both from the standpoint of compiler optimization and the language specification. The rapid increase in interest for using C++ that we observe in the scientific computing community implies that an opportunity to gain a strong foothold in this market is available. We consider that the many advantages for software development obtained by C++ are worth the price today but that continued rapid development of the fundamental technology with respect to operator overloading of numeric object types is essential to compete effectively in the future.

Acknowledgments

This work performed at Sandia National Laboratories supported by the U. S. Department of Energy under contract number DE-AC04-76DP00789.

References

[1] I. G. Angus and W. T. Thompkins, "Data Storage, Concurrency, and Portability: An Object Oriented Approach to Fluid Mechanics," The Fourth Conference on Hypercubes, Concurrent Computers and Applications, 1989.

[2] R. J. Collins, "CM++: A C++ Interface to the Connection Machine," Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications, Marist College, Sept. 1990.

[3] D. J. Miller and R. C. Lennox, "An Object-Oriented Environment for Robot System Architectures," IEEE International Conference on Robotics and Automation, 1990.

[4] D. W. Forslund, C. Wingate, P. Ford, J. S. Junkins, J. Jackson, S. C. Pope, "Experiences in Writing a Distributed Particle Simulation Code in C++," USENIX C++ Conference Proceedings, San Francisco, CA, April 9-11, 1990.

[5] R. J. Collins and D. R. Jefferson, "Selection in Massively Parallel Genetic Algorithms," Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann, 1991.

[6] A. Baden, C. Day, R. Grossman, D. Lifka, E. Lusk, E. May, And L. Price, "A data model for computations in high energy physics (preliminary report)," Laboratory for Advanced Computing Technical Report Number LAC91-R8, Univ. of Illinois at Chicago, December, 1991.

[7] C. M. Chase, A. L. Cheung, A. P. Reeves and M. R. Smith, "Paragon: A Parallel Programming Environment for Scientific Applications Using Communication Structures," 1991 International Conference on Parallel Processing.

[8] T. Keffer, "Object-Oriented Numerics, Part 1: Vectors, Matrices and All That Stuff," The C++ Journal, 1(4), 1991, pp. 3-9.

[9] I. G. Angus, "Parallelism, Object Oriented Programming Methods, Portable Software and C++," Proc. 8th Computing in Civil Engineering Symposium, American Society of Civil Engineers, 1992, pp. 506-513.

[10] D. W. Forslund, C. Wingate, P. Ford, J. Stephen Junkins, and S. C. Pope, “A Distributed Particle Simulation Code in C++,” Proc. 8th Computing in Civil Engineering Symposium, American Society of Civil Engineers, 1992, pp. 514-518.

[11] A. C. Robinson, A. L. Ames, H. Eliot, Fang, D. Pavlakos, C. T. Vaughan, and P. Campbell, “Massively Parallel Computing, C++ and Hydrocode Algorithms,” Proc. 8th Computing in Civil Engineering Symposium, American Society of Civil Engineers, 1992, pp. 519-526.

[12] J. S. Peery and K. G. Budge, “Experiences in Using C++ to Develop a Next Generation Strong Shock Wave Physics Code,” Proc. 8th Computing in Civil Engineering Symposium, American Society of Civil Engineers, 1992.

[13] T. J. Ross, J. P. Morrow, L. R. Wagner and G. F. Luger, “Two Paradigms for OOP Models for Scientific Applications,” Proc. 8th Computing in Civil Engineering Symposium, American Society of Civil Engineers, 1992, pp. 535-542.

[14] T. J. Ross, L. R. Wagner and G. F. Luger, “Object-Oriented programming for scientific codes: Thoughts and Concepts,” and “Object-Oriented programming for scientific codes: Examples in C++,” Univ. New Mexico Technical Report No. CS92-2, to appear in ASCE Journal of Computing in Civil Engineering.

[15] J. M. Coggins, “C++ in Numerical and Scientific Computing,” C++ Report, 4(3), 1992, pp. 65-68.

[16] M. B. Stephenson, S. A. Canann and T. D. Blacker, “Plastering: A New Approach to Automated, 3D Hexahedral Mesh Generation”, Sandia National Laboratories Report, SAND89-2192, February 1992.

[17] T. Keffer, “Object-Oriented Numerics, Part 2: Virtual Algorithms,” The C++ Journal, 2(2), 1992, pp. 3-8.

[18] I. G. Angus, “An Object Oriented Approach to Boundary Conditions in Finite Difference Fluid Dynamics Codes,” Scalable High Performance Computing Conference, 1992.

[19] I. G. Angus, “Image Algebra: An Object Oriented Approach to Transparently Concurrent Image Processing,” Scalable High Performance Computing Conference, 1992.

[20] D. Quinlan, “Workshop on C++ for Scientific Computing”, Abstracts in the proceedings of the SIAM Copper Mountain Conference on Iterative Methods, Copper Mountain, CO, April 9-14, 1992.

[21] W. S. Brainerd, C. H. Goldberg and J. C. Adams, *Programmer’s Guide to Fortran 90*, McGraw-Hill, 1990.

[22] L.M. Taylor and D.P. Flanagan, “PRONTO-3D: A Three-Dimensional Transient Solid Dynamics Program,” Sandia National Laboratories Report, SAND87-1912, 1989.

[23] K.G. Budge, “PHYSLIB: A C++ Tensor Class Library,” Sandia National Laboratories Report, SAND91-1752, 1991.

[24] G. Walker, “Why the Choice Must Be C++,” The C++ Journal 2(1), 1992.

[25] R. B. Davies, “Notes for the library working group of WG21/X3J16,” Presented at C++ Standards Committee Meeting, March 1991.

[26] N. M. Wilkinson, “C++ Return Value Optimization,” The C++ Journal, 2 (1), 1992, p. 47.